

# Intensionality, Intensional Recursion, and the Gödel-Löb axiom

G. A. Kavvos

Department of Computer Science  
University of Oxford  
Oxford, United Kingdom  
alex.kavvos@cs.ox.ac.uk

The use of a necessity-like modality in a typed  $\lambda$ -calculus can be used as a device for separating the calculus in two separate regions. These can be thought of as intensional vs. extensional data: data in the first region, the modal one, are available as code, and their description can be examined, whereas data in the second region are only available as values up to ordinary equality. This allows us to add seemingly non-functional operations at modal types, whilst maintaining consistency. In this setting the Gödel-Löb axiom acquires a novel constructive reading: it affords the programmer the possibility of a very strong kind of recursion, by enabling him to write programs that have access to their own code. This is a type of computational reflection that is strongly reminiscent of Kleene's Second Recursion Theorem. We prove that it is consistent with the rest of the system.

## 1 Introduction

This paper is about putting a logical twist on two old pieces of programming lore:

- First, it is about using *modal types* to treat *programs-as-data* in a type-safe manner.
- Second, it is about noticing that—in the context of intensional programming—a constructive reading of the Gödel-Löb axiom, i.e.  $\Box(\Box A \rightarrow A) \rightarrow \Box A$ , amounts to a strange kind of recursion, namely *intensional recursion*.

We will introduce a *typed  $\lambda$ -calculus with modal types* that supports both of these features. We will call it *Intensional PCF*, after the simply-typed  $\lambda$ -calculus with fixed points studied by Scott [21] and Plotkin [20].

### 1.1 Intensionality and Programs-as-data

To begin, we want to discuss our notion of *programs-as-data*. We mean it in a way that is considerably stronger than the higher-order functional programming with which we are already familiar, i.e. ‘functions as first-class citizens.’ In addition to that, our notion hints at a kind of *homoiconicity*, similar to the one present in the LISP family of languages. It refers to the ability given to a programmer to *quote* code, and carry it around as a datum; see [4] for an instance of that in LISP. This ability can be used for *metaprogramming*, which is the activity of writing programs that write other programs; indeed, this is what LISP macros excel at [10], and this is what the metaprogramming community has been studying for a long time: see [24, 26].

But we want to go even further. In LISP a program is able to process code by treating it as mere symbols, thereby disregarding its function and behaviour. This is what we call *intensionality*: an operation is *intensional* if it is *finer than equality*. Hence, it amounts to a kind of *non-functional behaviour*.

To our knowledge, this paper presents the first sound, type-safe attempt at intensional programming.

## 1.2 Intensional Recursion

We also want to briefly explain what we mean by *intensional recursion*—a fuller discussion may be found in [13, 1]. Most modern programming languages support *extensional recursion*: in the body of a function definition, the programmer may freely make a finite number of calls to the definiendum itself. Operationally, this leads a function to examine its own values at a finite set of points at which it has—hopefully—already been defined. In the *untyped  $\lambda$ -calculus*, this is modelled by the *First Recursion Theorem (FRT)* [3, §2.1, §6.1]:

**Theorem 1** (First Recursion Theorem).  $\forall f \in \Lambda. \exists u \in \Lambda. u = fu$ .

However, as Abramsky [1] notes, in the *intensional paradigm* we have described above, a stronger kind of recursion is imaginable. Instead of merely examining the result of a finite number of recursive calls, the definiendum can recursively have access to a *full copy of its own source code*. This is embodied in Kleene’s *Second Recursion Theorem (SRT)* [16]. Here is a version of the SRT in the untyped  $\lambda$ -calculus, where  $\llbracket u \rrbracket$  means ‘the Gödel number of the term  $u$ ’ [3, §6.5].

**Theorem 2** (Second Recursion Theorem).  $\forall f \in \Lambda. \exists u \in \Lambda. u = f \llbracket u \rrbracket$ .

Kleene also proved the following, where  $\Lambda^0$  is the set of closed  $\lambda$ -terms:

**Theorem 3** (Existence of Interpreter).  $\exists E \in \Lambda^0. \forall M \in \Lambda^0. E \llbracket M \rrbracket \rightarrow M$

As an interpreter exists, it is not hard to see that the SRT implies the FRT. It is not at all evident whether we can go the other way around. This is because the SRT is a *first-order theorem* that is about diagonalization and code, whereas the FRT really is about *higher types*. See the discussion in [13].

It thus follows that—in the presence of intensional operations—the SRT affords us with a much stronger kind of recursion. In fact, it allows for a certain kind of *computational reflection*, or *reflective programming*, of the same kind envisaged by Brian Cantwell Smith [22]. But the programme of Smith’s *reflective tower* involved a rather mysterious construction with unclear semantics [8, 29, 6], eventually leading to a theorem that—even in the presence of a mild reflective construct, the so-called **fexpr**—observational equivalence of programs collapses to  $\alpha$ -conversion [28].

We will use modalities to stop intension from flowing back into extension, so the theorem of [29] (which involves unrestricted quoting) will not apply. We will achieve reflection by internalising the SRT, and the key observation for doing so is the following. Suppose that our terms are typed, and that  $u : A$ . Suppose as well that there is a type constructor,  $\Box$ , so that  $\Box A$  means ‘code of type  $A$ .’ Then certainly  $\llbracket u \rrbracket : \Box A$ , and  $f$  is forced to have type  $\Box A \rightarrow A$ . A logical reading of the SRT is then the following: for every  $f : \Box A \rightarrow A$ , there exists a  $u : A$  such that  $u = f \llbracket f \rrbracket$ . This corresponds to Löb’s rule from *provability logic* [5]:

$$\frac{\Box A \rightarrow A}{A}$$

which is equivalent to adding the Gödel-Löb axiom to the logic. In fact, the punchline of this paper is that *the type of the Second Recursion Theorem is the Gödel-Löb axiom of provability logic*.

To obtain reflective features, then, we will add a version of Löb’s rule to our modal  $\lambda$ -calculus. It will be equivalent to the above, but proof-theoretically well-behaved—see [27, 14]:

$$\frac{\Box A \rightarrow A}{\Box A}$$

To our knowledge, this paper presents the first sound, type-safe attempt at reflective programming.

### 1.3 Prospectus

In §2 we will introduce the syntax of iPCF, and in §3 we will show that it satisfies basic metatheoretic properties. Follows that, in section §4, we show how to add intensional operations to iPCF. By proving that the resulting notion of reduction is confluent, we obtain consistency for the system. We then look at the computational behaviour of some important terms in §5, and conclude with two key examples of the new powerful features of our language in §6.

## 2 Introducing Intensional PCF

Intensional PCF (iPCF) is a typed  $\lambda$ -calculus with *modal types*. As discussed before, the modal types work in our favour by separating intension from extension, so that the latter does not leak into the former. Given the logical nature of the categorical constructs used in our previous work on intensionality [15], we shall model the types of iPCF after the *constructive modal logic* *S4*, in the dual-context style pioneered by Pfenning and Davies [19, 7]. Let us seize this opportunity to remark that (a) there are also other ways to capture *S4*, for which see the survey [13], and that (b) dual-context formulations are not by any means limited to *S4*: they began in the context of *intuitionistic linear logic*, but have recently been shown to also encompass other modal logics; see [14].

iPCF is *not* related to the language Mini-ML that is introduced by [7]: that is a call-by-value, ML-like language, with ordinary call-by-value fixpoints. In contrast, ours is a call-by-name language, with new kind of fixpoints, namely intensional fixpoints. These fixpoints will afford the programmer the full power of *intensional recursion*. Logically, they correspond to throwing the Gödel-Löb axiom, namely  $\Box(\Box A \rightarrow A) \rightarrow \Box A$  into *S4*. Modal logicians might object to this, as—in conjunction with the *T* axiom  $\Box A \rightarrow A$ —it will make every type inhabited. We remind them that a similar situation occurs in PCF, where the  $Y_A : (A \rightarrow A) \rightarrow A$  combinator allows one to write a term  $Y_A(\lambda x:A. x) : A$  for any  $A$ . As in the study of PCF, we mostly care about the underlying computation, so *it is the morphisms that matter, not the objects*.

The syntax and the typing rules of iPCF may be found in Figure 1. These are largely the same as Pfenning and Davies’ *DS4*, save the addition of some constants (drawn from PCF), and a rule for intensional recursion. The introduction rule for the modality restricts terms under a box ( $\Box$ ) to those containing only modal variables, i.e. variables carrying only intensions or code, but never ‘live values.’

$$\frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \text{box } M : \Box A}$$

There is also a rule for intensional recursion:

$$\frac{\Delta; z : \Box A \vdash M : A}{\Delta; \Gamma \vdash \text{fix } z \text{ in box } M : \Box A}$$

This will be coupled the reduction  $\text{fix } z \text{ in box } M \longrightarrow \text{box } M[\text{fix } z \text{ in box } M/z]$ . This rule is actually a version of *Löb’s rule*, and including it in the Hilbert system of a (classical or intuitionistic) modal logic is equivalent to including the Gödel-Löb axiom: see [5] and [27]. In fact, we have derived this fixpoint rule through general considerations of sequent calculi for GL; see [14]. Finally, let us record a fact noticed by Samson Abramsky, which is that erasing the modality from the types appearing in either Löb’s rule or the Gödel-Löb axiom yields the type of  $Y_A : (A \rightarrow A) \rightarrow A$ , either as a rule, or axiomatically internalised as a constant (both variants exist in the literature, see [11] and [18].)

Figure 1: Syntax and Typing Rules for Intensional PCF

<b>Ground Types</b>	$G$	$::=$	$\text{Nat} \mid \text{Bool}$
<b>Types</b>	$A, B$	$::=$	$G \mid A \rightarrow B \mid \Box A$
<b>Terms</b>	$M, N$	$::=$	$x \mid \lambda x:A. M \mid MN \mid \text{box } M \mid \text{let box } u \Leftarrow M \text{ in } N \mid$ $\hat{n} \mid \text{true} \mid \text{false} \mid \text{succ} \mid \text{pred} \mid \text{zero?} \mid \supset_G \mid \text{fix } z \text{ in box } M$
<b>Canonical Forms</b>	$V$	$::=$	$\hat{n} \mid \text{true} \mid \text{false} \mid \lambda x:A. M \mid \text{box } M$
<b>Contexts</b>	$\Gamma, \Delta$	$::=$	$\cdot \mid \Gamma, x:A$
<hr/>			
	$\Delta; \Gamma \vdash \hat{n} : \text{Nat}$		$\Delta; \Gamma \vdash b : \text{Bool} \quad (b \in \{\text{true}, \text{false}\})$
	$\Delta; \Gamma \vdash \text{zero?} : \text{Nat} \rightarrow \text{Bool}$		$\Delta; \Gamma \vdash f : \text{Nat} \rightarrow \text{Nat} \quad (f \in \{\text{succ}, \text{pred}\})$
	$\Delta; \Gamma \vdash \supset_G : \text{Bool} \rightarrow G \rightarrow G \rightarrow G$		
	$\Delta; \Gamma, x:A, \Gamma' \vdash x : A \quad (\text{var})$		$\Delta, u:A, \Delta'; \Gamma \vdash u : A \quad (\Box \text{var})$
	$\Delta; \Gamma, x:A \vdash M : B$		$\Delta; \Gamma \vdash M : A \rightarrow B \quad \Delta; \Gamma \vdash N : A$
	$\Delta; \Gamma \vdash \lambda x:A. M : A \rightarrow B \quad (\rightarrow \mathcal{I})$		$\Delta; \Gamma \vdash MN : B \quad (\rightarrow \mathcal{E})$
	$\Delta; \cdot \vdash M : A$		$\Delta; \Gamma \vdash M : \Box A \quad \Delta, u:A; \Gamma \vdash N : C$
	$\Delta; \Gamma \vdash \text{box } M : \Box A \quad (\Box \mathcal{I})$		$\Delta; \Gamma \vdash \text{let box } u \Leftarrow M \text{ in } N : C \quad (\Box \mathcal{E})$
	$\Delta; z : \Box A \vdash M : A$		
	$\Delta; \Gamma \vdash \text{fix } z \text{ in box } M : \Box A$		

### 3 Metatheory

iPCF satisfies the expected basic metatheoretic properties, namely structural and cut rules are admissible. This is no surprise given its origin in the well behaved Davies-Pfenning DS4. We assume the typical conventions for  $\lambda$ -calculi: terms are identified up to  $\alpha$ -equivalence, for which we write  $\equiv$ , and substitution  $[\cdot/\cdot]$  is defined in the ordinary, capture-avoiding manner. Bear in mind that we consider occurrences of  $u$  in  $N$  to be bound in  $\text{let box } u \Leftarrow M \text{ in } N$ . Contexts  $\Gamma, \Delta$  are lists of type assignments  $x : A$ . Furthermore, we shall assume that whenever we write a judgment like  $\Delta; \Gamma \vdash M : A$ , then  $\Delta$  and  $\Gamma$  are *disjoint*, in the sense that  $\text{VARS}(\Delta) \cap \text{VARS}(\Gamma) = \emptyset$ , where  $\text{VARS}(x_1 : A_1, \dots, x_n : A_n) \stackrel{\text{def}}{=} \{x_1, \dots, x_n\}$ . We write  $\Gamma, \Gamma'$  for the concatenation of disjoint contexts. Finally, we sometimes write  $\vdash M : A$  whenever  $\cdot; \cdot \vdash M : A$ .

**Theorem 4** (Structural & Cut). *The following rules are admissible in the typing system of iPCF:*

1. (Weakening)

$$\frac{\Delta; \Gamma, \Gamma' \vdash M : A}{\Delta; \Gamma, x:A, \Gamma' \vdash M : A}$$

3. (Contraction)

$$\frac{\Delta; \Gamma, x:A, y:A, \Gamma' \vdash M : A}{\Delta; \Gamma, w:A, \Gamma' \vdash M[w, w/x, y] : A}$$

2. (Exchange)

$$\frac{\Delta; \Gamma, x:A, y:B, \Gamma' \vdash M : C}{\Delta; \Gamma, y:B, x:A, \Gamma' \vdash M : C}$$

4. (Cut)

$$\frac{\Delta; \Gamma \vdash N : A \quad \Delta; \Gamma, x:A, \Gamma' \vdash M : A}{\Delta; \Gamma, \Gamma' \vdash M[N/x] : A}$$

**Theorem 5** (Modal Structural & Cut). *The following rules are admissible:*

1. (Modal Weakening)

$$\frac{\Delta, \Delta'; \Gamma \vdash M : C}{\Delta, u:A, \Delta'; \Gamma \vdash M : C}$$

3. (Modal Contraction)

$$\frac{\Delta, x:A, y:A, \Delta'; \Gamma \vdash M : C}{\Delta, w:A, \Delta'; \Gamma \vdash M[w, w/x, y] : C}$$

2. (Modal Exchange)

$$\frac{\Delta, x:A, y:B, \Delta'; \Gamma \vdash M : C}{\Delta, y:B, x:A, \Delta'; \Gamma \vdash M : C}$$

4. (Modal Cut)

$$\frac{\Delta; \cdot \vdash N : A \quad \Delta, u:A, \Delta'; \Gamma \vdash M : C}{\Delta, \Delta'; \Gamma \vdash M[N/u] : C}$$

### 4 Consistency of Intensional Operations

In this section we shall prove that the modal types of iPCF allow us to consistently add ‘intensional operations’ on the modal types. These are *non-functional operations on terms*, which are not ordinarily definable, because they violate equality. However, we will show that our use of modal types enables the separation of intension from extension. We shall do this by adding intensional operations to iPCF, introducing a notion of reduction, and proving it confluent. A known corollary of confluence is that the equational theory induced by reduction is *consistent*, i.e. it does not equate all terms.

There is one very serious caveat, involving extension flowing into intension. That is, we need to exclude from consideration terms where a variable bound by a  $\lambda$  occurs under the scope of a box  $(-)$  construct. These will never be well-typed, but—since we discuss types and reduction orthogonally—we also need to explicitly exclude them here too.

#### 4.1 Adding intensionality

Davies and Pfenning [19] suggested that the modal type can be used to signify intensionality. In fact, in their previous paper [7] they had prevented reductions from happening under  $\text{box } (-)$  construct, “[...] since this would violate its intensional nature.” But the truth is that neither of these presentations included any genuinely *non-functional operations* at modal types, and hence their only use was for staged metaprogramming.

Adding intensional, non-functional operations is a much more difficult task. Intensional operations are dependent on *descriptions* and *intensions* rather than *values* and *extensions*. Hence, unlike reduction and evaluation, they cannot be blind to substitution. This is the most challenging aspect to the whole endeavour.

The task was recently taken up by Gabbay and Nanevski [9], who attempted to add a construct  $\text{is-app}$  to the system of Davies and Pfenning, with the reduction rules

$$\begin{aligned} \text{is-app } (PQ) &\longrightarrow \text{true} \\ \text{is-app } M &\longrightarrow \text{false} \quad \text{if } M \text{ is not an application term} \end{aligned}$$

They tried to justify this addition in terms of a denotational semantics for modal types, in which the semantic domain  $\llbracket \Box A \rrbracket$  directly involves the actual closed syntax of type  $\Box A$ . But something seems to have gone wrong with substitution. In fact, we believe that their proof of soundness is wrong: it is not hard to see that their semantics is not stable under the second of those two reductions: take  $M$  to be  $u$ , and let the semantic environment map  $u$  to an application  $\text{box } PQ$ . We can also see this in the fact that their notion of reduction is *not confluent*; here is the relevant counterexample: we can reduce

$$\text{let box } u \Leftarrow \text{box } (PQ) \text{ in is-app } (\text{box } u) \longrightarrow \text{is-app } (\text{box } PQ) \longrightarrow \text{true}$$

but we can also reduce as follows:

$$\text{let box } u \Leftarrow \text{box } (PQ) \text{ in is-app } (\text{box } u) \longrightarrow \text{let box } u \Leftarrow \text{box } (PQ) \text{ in false} \longrightarrow \text{false}$$

This is easily discoverable if one tries to plough through a proof of confluence: it is very clearly *not* the case that  $M \rightarrow N$  implies  $M[P/u] \rightarrow N[P/u]$  if  $u$  is under a  $\text{box } (-)$  and operations made of the same ilk as  $\text{is-app}$ . Perhaps the following idea has the makings of a workable proposal: let us limit intensional operations to a chosen set of functions  $f : \mathcal{T}(A) \rightarrow \mathcal{T}(B)$  from terms of type  $A$  to terms of type  $B$ , and then represent them in the language by a constant  $\tilde{f}$ , such that  $\tilde{f}(\text{box } M) \rightarrow \text{box } f(M)$ . This set of functions would then be chosen so that they satisfy some sanity conditions. Since we want to have a  $\text{let}$  construct that allows us to substitute code variables for code, the following general situation will occur: if  $N \rightarrow N'$ , we have

$$\text{let box } u \Leftarrow \text{box } M \text{ in } N \longrightarrow N[M/u]$$

and

$$\text{let box } u \Leftarrow \text{box } M \text{ in } N \longrightarrow \text{let box } u \Leftarrow \text{box } M \text{ in } N' \longrightarrow N'[M/u]$$

Thus, in order to have confluence, we need  $N[M/u] \rightarrow N'[M/u]$ . This will only be the case for reductions of the form  $\tilde{f}(\text{box } M) \rightarrow \text{box } f(M)$  if  $f(N[M/u]) \equiv (f(N)) [M/u]$ , i.e. if  $f$  is *substitutive*. But then a simple naturality argument gives that  $f(N) \equiv f(u[N/u]) \equiv (f(u)) [N/u]$ , and hence  $\tilde{f}$  is already definable by  $\lambda x:\Box A. \text{let box } u \Leftarrow x \text{ in box } f(u)$ , so such a ‘substitutive’ function is neither intensional nor non-functional after all.

In fact, the only truly intensional operations we can add to our calculus will be those acting on *closed* terms. We will see that this circumvents the problems that arise when intensionality interacts with substitution. So we will limit intensional operations to the following set:

**Definition 1** (Intensional operations). Let  $\mathcal{T}(A)$  be the set of terms  $M$  such that  $\cdot; \cdot \vdash M : A$ . Let  $\mathcal{F}(A, B)$  be the set of all functions  $f : \mathcal{T}(A) \rightarrow \mathcal{T}(B)$ .

We will include all of these intensional operations  $f : \mathcal{T}(A) \rightarrow \mathcal{T}(B)$  in our calculus, as constants:

$$\Delta; \Gamma \vdash \tilde{f} : \Box A \rightarrow \Box B$$

with reduction rule  $\tilde{f}(\Box M) \rightarrow \Box f(M)$ , under the proviso that  $M$  is closed. Of course, this also includes operations on terms that might in some sense *not* be computable. However, we are interested in proving consistency in the most general setting. The questions of which intensional operations are computable, and which primitives can and should be used to express them, are both still open.

## 4.2 Reduction and Confluence

We introduce a notion of  $\beta$ -reduction for iPCF, which we present in Figure 2. Unlike most studies of PCF-like languages, we do not consider a reduction strategy at first, but simply ordinary, ‘non-deterministic’  $\beta$ -reduction. We do so because we are trying to show consistency.

The equational theory induced by this notion of reduction is a symmetric version of it, annotated with types. It is easy to write down, so we omit it. Note that fact that, like in the work of Davies and Pfenning, we do *not* include the congruence rule for the modality:

$$\frac{\Delta; \cdot \vdash M = N : A}{\Delta; \Gamma \vdash \Box M = \Box N : \Box A} (\Box \text{cong})$$

In fact, the very absence of this rule is what will allow modal types to become intensional. Otherwise, the only new rules are (a) intensional recursion, embodied in the rule ( $\Box \text{fix}$ ), and (b) intensional operations, exemplified by the rule ( $\Box \text{int}$ ).

We can now show that

**Theorem 6.** *The reduction relation  $\longrightarrow$  is confluent.*

The easiest route to that theorem is to use a proof like that in [14], i.e. the method of *parallel reduction*. This kind of proof was originally discovered by Tait and Martin-Löf, and is nicely documented in [25]. Because of the intensional nature of our  $\Box$  ( $-$ ) constructs, ours will be more nuanced and fiddly. We omit the details, for want of space (but see the appendix).

## 5 Some important terms

Let us look at the kinds of terms we can write in iPCF.

**From the axioms of S4** First, we can write a term corresponding to axiom K, the *normality axiom*:

$$\text{ax}_K \stackrel{\text{def}}{=} \lambda f : \Box(A \rightarrow B). \lambda x : \Box A. \text{let } \Box g \Leftarrow f \text{ in let } \Box y \Leftarrow x \text{ in } \Box(g y)$$

It is easy to see that  $\vdash \text{ax}_K : \Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$ . An intensional reading of this is as follows: any function given as code can be transformed into an *effective operation*, mapping code to code.

Figure 2: Beta reduction for Intensional PCF

$$\begin{array}{c}
\frac{}{(\lambda x:A. M)N \longrightarrow M[N/x]} (\longrightarrow \beta) \quad \frac{M \longrightarrow N}{\lambda x:A. M \longrightarrow \lambda x:A. N} (\text{cong}_\lambda) \\
\\
\frac{M \longrightarrow N}{MP \longrightarrow NP} (\text{app}_1) \quad \frac{P \longrightarrow Q}{MP \longrightarrow MQ} (\text{app}_2) \\
\\
\frac{}{\text{let box } u \Leftarrow \text{box } M \text{ in } N \longrightarrow N[M/u]} (\Box\beta) \\
\\
\frac{}{\text{fix } z \text{ in box } M \longrightarrow \text{box } M[\text{fix } z \text{ in box } M/z]} (\Box\text{fix}) \\
\\
\frac{M \text{ closed, } f(M) \downarrow}{\tilde{f}(\text{box } M) \longrightarrow \text{box } f(M)} (\Box\text{int}) \\
\\
\frac{M \longrightarrow N}{\text{let box } u \Leftarrow M \text{ in } P \longrightarrow \text{let box } u \Leftarrow N \text{ in } P} (\text{let-cong}_1) \\
\\
\frac{P \longrightarrow Q}{\text{let box } u \Leftarrow M \text{ in } P \longrightarrow \text{let box } u \Leftarrow M \text{ in } Q} (\text{let-cong}_2) \\
\\
\frac{}{\text{zero? } \widehat{0} \longrightarrow \text{true}} (\text{zero?}_1) \quad \frac{}{\text{zero? } \widehat{n+1} \longrightarrow \text{false}} (\text{zero?}_2) \\
\\
\frac{}{\text{succ } \widehat{n} \longrightarrow \widehat{n+1}} (\text{succ}) \quad \frac{}{\text{pred } \widehat{n} \longrightarrow \widehat{n-1}} (\text{pred}) \\
\\
\frac{}{\supset_G \text{ true } M N \longrightarrow M} (\supset_1) \quad \frac{}{\supset_G \text{ false } M N \longrightarrow N} (\supset_2)
\end{array}$$



The rest of the axioms correspond to evaluating and quoting, as discussed before. Axiom T takes code to value, or intension to extension:

$$\vdash \text{eval}_A \stackrel{\text{def}}{=} \lambda x : \Box A. \text{let box } y \Leftarrow x \text{ in } y : \Box A \rightarrow A$$

and axiom 4 quotes code into code-for-code:

$$\vdash \text{quote}_A \stackrel{\text{def}}{=} \lambda x : \Box A. \text{let box } y \Leftarrow x \text{ in box } (\text{box } y) : \Box A \rightarrow \Box \Box A$$

**The Gödel-Löb axiom: intensional fixed points** Since the  $(\Box \text{fix})$  rule is a variant of Löb's rule, we expect to already have a term corresponding to the Gödel-Löb axiom of provability logic. We do—and it is an *intensional fixed-point combinator*:

$$\mathbb{Y}_A \stackrel{\text{def}}{=} \lambda x : \Box(\Box A \rightarrow A). \text{let box } f \Leftarrow x \text{ in } (\text{fix } z \text{ in box } (f z))$$

and  $\vdash \mathbb{Y}_A : \Box(\Box \rightarrow A) \rightarrow \Box A$ . We observe that

$$\mathbb{Y}_A(\text{box } M) \longrightarrow^* \text{fix } z \text{ in box } M z \longrightarrow \text{box } M(\text{fix } z \text{ in box } M z)$$

**Undefined** The combination of eval and  $\mathbb{Y}_A$  lead to non-termination, in a style reminiscent of the term  $(\lambda x.xx)(\lambda x.xx)$  of the untyped  $\lambda$ -calculus. Let

$$\Omega_A \stackrel{\text{def}}{=} \text{fix } z \text{ in box } (\text{eval}_A z)$$

Then  $\vdash \Omega_A : \Box A$ , and  $\Omega_A \longrightarrow \text{box } (\text{eval}_A \Omega_A)$ , so that  $\text{eval}_A \Omega_A \longrightarrow^* \text{eval}_A \Omega_A$ .

**Extensional Fixed Points** Perhaps surprisingly, the ordinary PCF **Y** combinator is also definable in the iPCF. Let

$$U_A \stackrel{\text{def}}{=} \text{fix } z \text{ in box } (\lambda f : A \rightarrow A. f(\text{eval } z f))$$

Then  $\vdash U_A : \Box((A \rightarrow A) \rightarrow A)$ , and we can define

$$\mathbf{Y}_A \stackrel{\text{def}}{=} \text{eval}_{(A \rightarrow A) \rightarrow A} U_A$$

with the result that  $\mathbf{Y}_A f \longrightarrow^* f(\mathbf{Y}_A f)$ .

## 6 Two intensional examples

No description of an intensional language with intensional recursion would be complete without an example of both of these features. Our first example is about intensionality, and is drawn from the study of PCF and issues related to sequential vs. parallel (but not concurrent) computation. Our second example is slightly more adventurous: it is a computer virus.

## 6.1 ‘Parallel or’ by dovetailing

In [20], Gordon Plotkin proved the following theorem: there is no term  $\text{por} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$  of PCF such that  $\text{por true } M \rightarrow_{\beta} \text{true}$  and  $\text{por } M \text{ true} \rightarrow_{\beta} \text{true}$  for any  $\vdash M : \text{Bool}$ , whereas  $\text{por false false} \rightarrow_{\beta} \text{false}$ . Intuitively, the problem is that  $\text{por}$  has to first examine one of its two arguments, and this can be troublesome if that argument is non-terminating. It follows that the *parallel or* function is not definable in PCF. In order to regain the property of so-called *full abstraction* for the *Scott model* of PCF, a constant denoting this function has to be manually added to PCF, and endowed with the above rather clunky operational semantics. See [20, 11, 18, 23].

However, the *parallel or* function is a perfectly computable *partial recursive functional* [23, 17]. The way to prove that is informally the following: given two closed terms  $M, N : \text{Bool}$ , take turns in  $\beta$ -reducing each one for a few steps. This is called *dovetailing*. If at any point one of the two terms reduces to true, then output true. But if at any point both reduce to false then output false.

This procedure is not definable in PCF, because a candidate term  $\text{por}$  does not have access to a code for its argument, for it can only inspect its value. But in iPCF, we can use the modality to have access to code, and intensional operations to implement reduction. Suppose we pick a reduction strategy  $\rightarrow_r$ . Then, let us include a constant  $\text{tick} : \Box\text{Bool} \rightarrow \Box\text{Bool}$  that implements one step of this reduction strategy on closed terms:

$$\frac{M \rightarrow_r N}{\text{tick}(\text{box } M) \rightarrow \text{box } N}$$

Also, let us include a constant  $\text{done?} : \Box\text{Bool} \rightarrow \text{Bool}$ , which tells us if a closed term under a box is a normal form:

$$\frac{M \text{ normal}}{\text{done?}(\text{box } M) \rightarrow \text{true}} \quad \frac{M \text{ not normal}}{\text{done?}(\text{box } M) \rightarrow \text{false}}$$

These two can be subsumed under our previous scheme for introducing intensional operations. The above argument is now implemented by the following term:

$$\begin{aligned} \text{por} \equiv & \mathbf{Y}(\lambda \text{por}. \lambda x : \Box\text{Bool}. \lambda y : \Box\text{Bool}. \\ & \supset_{\text{Bool}}(\text{done? } x) \quad (\text{lor } (\text{eval } x)(\text{eval } y)) \\ & \quad \quad \quad (\supset_{\text{Bool}}(\text{done? } y) \quad (\text{ror } (\text{eval } x)(\text{eval } y)) \\ & \quad \quad \quad (\text{por } (\text{tick } x)(\text{tick } y))) \end{aligned}$$

where  $\text{lor}, \text{ror} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$  are terms defining the left-strict and right-strict versions of the ‘or’ connective respectively. Notice that the type of this term of  $\Box\text{Bool} \rightarrow \Box\text{Bool} \rightarrow \text{Bool}$ : we require *intensional access* to the terms of boolean type in order to define this computation.

## 6.2 A computer virus

*Abstract computer virology* is the study of formalisms that model computer viruses. There are many ways to formalise viruses. We will use the model of Adleman [2], where files can be interpreted either as data, or as functions. We introduce a data type  $F$ , and two constants

$$\text{in} : \Box(F \rightarrow F) \rightarrow F \quad \text{and} \quad \text{out} : F \rightarrow \Box(F \rightarrow F)$$

such that  $\text{out}(\text{in } M) \rightarrow M$ , making  $\Box(F \rightarrow F)$  a retract of  $F$ . This might seem the same as the situation where  $F \rightarrow F$  is a retract of  $F$ , which yields models of the (untyped)  $\lambda$ -calculus and is difficult to

construct [3, §5.4], but it is not nearly as worrying:  $\Box(F \rightarrow F)$  is populated by intensions and codes, not by actual functions. Under this interpretation, the pair (in,out) corresponds to a kind of Gödel numbering—especially if  $F$  is  $\mathbb{N}$ .

Now, in Adleman’s model, a *virus* is given by its infected form, which either *injures*, *infects*, or *imitates* other programs. The details are unimportant in the present discussion, save from the fact that one can construct such a virus from its *infection routine* by using Kleene’s SRT. Let us model it by a term  $\text{infect} : \Box(F \rightarrow F) \rightarrow F \rightarrow F$ , which accepts a piece of viral code and a file, and it returns either the file itself, or a version infected with the viral code. We can then define a term  $\text{virus} : \Box(F \rightarrow F)$ :

$$\text{virus} \stackrel{\text{def}}{=} \text{fix } z \text{ in box infect } z$$

so that  $\text{virus} \longrightarrow^* \text{box (infect virus)}$ . We then have that

$$\text{eval virus} \longrightarrow^* \text{infect virus}$$

which is a program that is ready to infect its input with its own code.

## 7 Conclusion

We have achieved the desideratum of an intensional programming language, with intensional recursion. There are two main questions that result from this development.

Firstly, does there exist a good set of *intensional primitives* from which all others are definable? Is there perhaps *more than one such set*, hence providing us with a choice of programming primitives?

Secondly, what is the exact kind of programming power that we have unleashed? Does it lead to interesting programs that we have not been able to write before? We have outlined some speculative applications for intensional recursion in [12]. Is iPCF a useful tool when it comes to attacking these?

## Acknowledgments

I would like to thank Mario Alvarez-Picallo for our endless conversations on types and metaprogramming. This work was supported by the EPSRC as part of my doctoral research (award reference 1354534).

## References

- [1] Samson Abramsky (2014): *Intensionality, Definability and Computation*. In Alexandru Baltag & Sonja Smets, editors: *Johan van Benthem on Logic and Information Dynamics*, Springer-Verlag, pp. 121–142, doi:10.1007/978-3-319-06025-5\_5.
- [2] Leonard M Adleman (1990): *An Abstract Theory of Computer Viruses*. In: *Advances in Cryptology - CRYPTO’ 88, Lecture Notes in Computer Science* 403, Springer New York, New York, NY, pp. 354–374, doi:10.1007/0-387-34799-2\_28.
- [3] Henk Barendregt (1984): *Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam.
- [4] Alan Bawden (1999): *Quasiquotation in LISP*. In: *Proceedings of the 6th ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’99)*. Available at <http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf>.
- [5] George S. Boolos (1994): *The Logic of Provability*. Cambridge University Press, Cambridge, doi:10.1017/CBO9780511625183.

- [6] Olivier Danvy & Karoline Malmkjaer (1988): *Intensions and extensions in a reflective tower*. In: *Proceedings of the 1988 ACM conference on LISP and functional programming (LFP '88)*, ACM Press, New York, New York, USA, pp. 327–341, doi:10.1145/62678.62725.
- [7] Rowan Davies & Frank Pfenning (2001): *A modal analysis of staged computation*. *Journal of the ACM* 48(3), pp. 555–604, doi:10.1145/382780.382785.
- [8] Daniel P. Friedman & Mitchell Wand (1984): *Reification: Reflection without metaphysics*. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming (LFP '84)*, ACM Press, New York, New York, USA, pp. 348–355, doi:10.1145/800055.802051.
- [9] Murdoch J. Gabbay & Aleksandar Nanevski (2013): *Denotation of contextual modal type theory (CMTT): Syntax and meta-programming*. *Journal of Applied Logic* 11(1), pp. 1–29, doi:10.1016/j.jal.2012.07.002.
- [10] Paul Graham (1993): *On LISP: Advanced Techniques for Common LISP*. Prentice Hall.
- [11] Carl A Gunter (1992): *Semantics of programming languages: structures and techniques*. Foundations of Computing, The MIT Press.
- [12] G. A. Kavvos (2016): *Kleene's Two Kinds of Recursion*. CoRR. Available at <http://arxiv.org/abs/1602.06220>.
- [13] G. A. Kavvos (2016): *The Many Worlds of Modal  $\lambda$ -calculi: I. Curry-Howard for Necessity, Possibility and Time*. CoRR. Available at <http://arxiv.org/abs/1605.08106>.
- [14] G. A. Kavvos (2017): *Dual-Context Calculi for Modal Logic*. CoRR. Available at <http://arxiv.org/abs/1602.04860>.
- [15] G. A. Kavvos (2017): *On the Semantics of Intensionality*. In: *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. Available at <http://arxiv.org/abs/1602.01365>.
- [16] Stephen Cole Kleene (1938): *On notation for ordinal numbers*. *The Journal of Symbolic Logic* 3(04), pp. 150–155, doi:10.2307/2267778.
- [17] John R. Longley & Dag Normann (2015): *Higher-Order Computability*. Theory and Applications of Computability, Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-662-47992-6.
- [18] John C Mitchell (1996): *Foundations for programming languages*. Foundations of Computing, The MIT Press.
- [19] Frank Pfenning & Rowan Davies (2001): *A judgmental reconstruction of modal logic*. *Mathematical Structures in Computer Science* 11(4), pp. 511–540, doi:10.1017/S0960129501003322.
- [20] Gordon D. Plotkin (1977): *LCF considered as a programming language*. *Theoretical Computer Science* 5(3), pp. 223–255, doi:10.1016/0304-3975(77)90044-5.
- [21] Dana S. Scott (1993): *A type-theoretical alternative to ISWIM, CUCH, OWHY*. *Theoretical Computer Science* 121(1-2), pp. 411–440, doi:10.1016/0304-3975(93)90095-B.
- [22] Brian Cantwell Smith (1984): *Reflection and Semantics in LISP*. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*, ACM Press, New York, New York, USA, pp. 23–35, doi:10.1145/800017.800513.
- [23] Thomas Streicher (2006): *Domain-theoretic Foundations of Functional Programming*. World Scientific.
- [24] Walid Taha & Tim Sheard (2000): *MetaML and multi-stage programming with explicit annotations*. *Theoretical Computer Science* 248(1-2), pp. 211–242, doi:10.1016/S0304-3975(00)00053-0.
- [25] M. Takahashi (1995): *Parallel Reductions in  $\lambda$ -Calculus*. *Information and Computation* 118(1), pp. 120–127, doi:10.1006/inco.1995.1057.
- [26] Takeshi Tsukada & Atsushi Igarashi (2010): *A logical foundation for environment classifiers*. *Logical Methods in Computer Science* 6(4), pp. 1–43, doi:10.2168/LMCS-6(4:8)2010.

- [27] Aldo Ursini (1979): *A modal calculus analogous to K4W, based on intuitionistic propositional logic*. *Studia Logica* 38(3), pp. 297–311, doi:10.1007/BF00405387. Available at <http://link.springer.com/10.1007/BF00405387>.
- [28] Mitchell Wand (1998): *The Theory of Fexprs is Trivial*. *LISP and Symbolic Computation* 10(3), pp. 189–199, doi:10.1023/A:1007720632734.
- [29] Mitchell Wand & Daniel P. Friedman (1988): *The mystery of the tower revealed: A nonreflective description of the reflective tower*. *Lisp and Symbolic Computation* 1(1), pp. 11–38, doi:10.1007/BF01806174.

## A Appendix: Proof of confluence

We will use a variant of the proof in [14], i.e. the method of *parallel reduction*. This kind of proof was originally discovered by Tait and Martin-Löf, and is nicely documented in [25]. Beause of the intensional nature of our box  $(-)$  constructs, ours will be more nuanced and fiddly than any in *op. cit.* The method largely follows the following pattern: we will introduce a second notion of reduction,

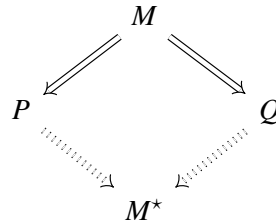
$$\Longrightarrow \subseteq \Lambda \times \Lambda$$

which we will ‘sandwich’ between reduction proper and its transitive closure:

$$\longrightarrow \subseteq \Longrightarrow \subseteq \longrightarrow^*$$

We will then show that  $\Longrightarrow$  has the diamond property. By the above inclusions, the transitive closure  $\Longrightarrow^*$  of  $\Longrightarrow$  is then equal to  $\longrightarrow^*$ , and hence  $\longrightarrow$  is Church-Rosser.

In fact, we will follow [25] in doing something better: we will define for each term  $M$  its *complete development*,  $M^*$ . The complete development is intuitively defined by ‘unrolling’ all the redexes of  $M$  at once. We will then show that if  $M \Longrightarrow N$ , then  $N \Longrightarrow M^*$ .  $M^*$  will then suffice to close the diamond:



The parallel reduction  $\Longrightarrow$  is defined in Figure 3. Ordinarily, instead of the axiom (refl) ensuring  $M \Longrightarrow M$ , we would simply have an axiom for variables,  $x \Longrightarrow x$ . Then,  $M \Longrightarrow M$  would be derivable. However, we do not have a congruence rule for box  $(-)$ , and also our fixpoint construct would block the possibility of this. We are thus forced to include  $M \Longrightarrow M$ , which slightly complicates the lemmas that follow.

Indeed, the main lemma that usually underpins the confluence proof is this: if  $M \Longrightarrow N$  and  $P \Longrightarrow Q$ ,  $M[P/x] \Longrightarrow N[Q/x]$ . However, this is intuitively wrong: no reductions should happen under boxes, so this should only hold if we are substituting for a variable *not* occurring under boxes. Hence, this lemma splits into three different ones:

- one showing that  $P \Longrightarrow Q$  implies  $M[P/x] \Longrightarrow M[Q/x]$ , if  $x$  does not occur under boxes: this is the price to pay for replacing the variable axiom with (refl);
- one showing that  $M \Longrightarrow N$  implies  $M[P/u] \Longrightarrow N[P/u]$ , even if  $u$  is under a box; and

Figure 3: Parallel Reduction

$$\begin{array}{c}
\frac{}{M \Longrightarrow M} \text{ (refl)} \\
\\
\frac{M \Longrightarrow N}{\lambda x:A. M \Longrightarrow \lambda x:A. N} \text{ (cong}_\lambda\text{)} \\
\\
\frac{M \Longrightarrow M'}{\supset_G \text{ true } M N \Longrightarrow M'} (\supset_1) \\
\\
\frac{M \Longrightarrow N \quad P \Longrightarrow Q}{(\lambda x:A. M)P \Longrightarrow N[Q/x]} (\rightarrow \beta) \\
\\
\frac{M \Longrightarrow N \quad P \Longrightarrow Q}{MP \Longrightarrow NQ} \text{ (app)} \\
\\
\frac{N \Longrightarrow N'}{\supset_G \text{ false } M N \Longrightarrow N'} (\supset_2) \\
\\
\frac{M \Longrightarrow N}{\text{let box } u \Leftarrow \text{box } P \text{ in } M \Longrightarrow N[P/u]} (\Box\beta) \\
\\
\frac{}{\text{fix } z \text{ in box } M \Longrightarrow \text{box } M[\text{fix } z \text{ in box } M/z]} (\Box\text{fix}) \\
\\
\frac{M \text{ closed, } f(M) \downarrow}{\tilde{f}(\text{box } M) \Longrightarrow \text{box } f(M)} (\Box\text{int}) \\
\\
\frac{M \Longrightarrow N \quad P \Longrightarrow Q}{\text{let box } u \Leftarrow M \text{ in } P \Longrightarrow \text{let box } u \Leftarrow N \text{ in } Q} (\Box\text{let-cong})
\end{array}$$

**Remark.** In addition to the above, one should also include rules for the constants, but these are merely restatements of the rules in Figure 2.

- one showing that, if  $x$  does not occur under boxes,  $M \Longrightarrow N$  and  $P \Longrightarrow Q$  indeed imply  $M[P/x] \Longrightarrow N[Q/x]$

But let us proceed with the proof.

**Lemma 1.** *If  $M \Longrightarrow N$  then  $M[P/u] \Longrightarrow N[P/u]$ .*

*Proof.* By induction on the generation of  $M \Longrightarrow N$ . Most cases trivially follow, or consist of simple invocations of the IH. In the case of  $(\rightarrow \beta)$ , the known substitution lemma suffices. Let us look at the cases involving boxes.

CASE( $\square\beta$ ). Then  $M \Longrightarrow N$  is let box  $v \Leftarrow$  box  $R$  in  $S \Longrightarrow S'[R/v]$  with  $S \Longrightarrow S'$ . By the IH, we have that  $S[P/u] \Longrightarrow S'[P/u]$ , so let box  $v \Leftarrow$  box  $R[P/u]$  in  $S[P/u] \Longrightarrow S[P/u][R[P/u]/v]$ , and this last is  $\alpha$ -equivalent to  $S[R/v][P/u]$  by the substitution lemma.

CASE( $\square\text{fix}$ ). A similar application of the substitution lemma.

CASE( $\square\text{int}$ ). Then  $M \Longrightarrow N$  is  $\tilde{f}(\text{box } Q) \Longrightarrow \text{box } f(Q)$ , with  $Q$  closed. Hence

$$(\tilde{f}(\text{box } Q)) [P/u] \equiv \tilde{f}(\text{box } Q) \Longrightarrow \text{box } f(Q) \equiv (f(Q)) [P/u]$$

simply because  $Q$  is closed.

□

**Lemma 2.** *If  $P \Longrightarrow Q$  and  $x \notin \text{FV}_{\geq 1}(M)$ , then  $M[P/x] \Longrightarrow M[Q/x]$ .*

*Proof.* By induction on the term  $M$ . The only non-trivial cases are those for  $M$  being a variable, or box  $M'$ . In the first case, depending on which variable  $M$  is, use either (refl), or the assumption  $P \Longrightarrow Q$ . In the second case,  $(\text{box } M')[P/x] \equiv \text{box } M' \equiv (\text{box } M')[Q/x]$  as  $x$  does not occur under a box, so use (refl) again. □

**Lemma 3.** *If  $M \Longrightarrow N$ ,  $P \Longrightarrow Q$ , and  $x \notin \text{FV}_{\geq 1}(M)$ , then*

$$M[P/x] \Longrightarrow N[Q/x]$$

*Proof.* By induction on the generation of  $M \Longrightarrow N$ . The cases for most congruence rules and constants follow trivially, or from the IH. We prove the rest.

CASE(refl). Then  $M \Longrightarrow N$  is actually  $M \Longrightarrow M$ , so we use Lemma 2 to infer  $M[P/x] \Longrightarrow M[Q/x]$ .

CASE( $\square\text{int}$ ). Then  $M \Longrightarrow N$  is actually  $\tilde{f}(\text{box } M) \Longrightarrow \text{box } f(M)$ . But  $M$  is then closed, so  $(\tilde{f}(\text{box } M)) [P/x] \equiv \tilde{f}(\text{box } M) \Longrightarrow \text{box } f(M) \equiv (\text{box } f(M)) [Q/x]$ .

CASE( $\supset_i$ ). Then  $M \Longrightarrow N$  is  $\supset_G \text{ true } M \ N \Longrightarrow M'$  with  $M \Longrightarrow M'$ . By the IH,  $M[P/x] \Longrightarrow M'[Q/x]$ , so

$$\supset_G \text{ true } M[P/x] \ N[P/x] \Longrightarrow M'[Q/x]$$

by a single use of ( $\supset_1$ ). The case for false is similar.

CASE( $\rightarrow \beta$ ). Then  $(\lambda x':A. M)N \Longrightarrow N'[M'/x']$ , where  $M \Longrightarrow M'$  and  $N \Longrightarrow N'$ . Then

$$((\lambda x':A. M)N) [P/x] \equiv (\lambda x':A. M[P/x])(N[P/x])$$

But, by the IH,  $M[P/x] \implies M'[Q/x]$  and  $N[P/x] \implies N'[Q/x]$ . So, by the rules ( $\text{cong}_\lambda$ ) and ( $\text{app}$ ), and then rule ( $\rightarrow \beta$ ), we have

$$(\lambda x'.A. M[P/x])(N[P/x]) \implies M'[Q/x] [N'[Q/x]/x']$$

But this last is  $\alpha$ -equivalent to  $(M'[N'/x'])[Q/x]$  by the substitution lemma.

CASE( $\Box\beta$ ). Then let  $\text{box } u' \Leftarrow \text{box } M$  in  $N \implies N'[M/u']$  where  $N \implies N'$ . By assumption, we have that  $x \notin \text{FV}(M)$  and  $x \notin \text{FV}_{\geq 1}(N)$ . Hence, we have by the IH that  $N[P/x] \implies N'[Q/x]$ , so by applying ( $\Box\beta$ ) we get

$$\begin{aligned} (\text{let box } u' \Leftarrow \text{box } M \text{ in } N)[P/x] &\equiv \text{let box } u' \Leftarrow \text{box } M[P/x] \text{ in } N[P/x] \\ &\equiv \text{let box } u' \Leftarrow \text{box } M \text{ in } N[P/x] \\ &\implies N'[Q/x][M/u'] \end{aligned}$$

But this last is  $\alpha$ -equivalent to  $N'[M/u'][Q/x]$ , by the substitution lemma and the fact that  $x$  does not occur in  $M$ .

CASE( $\Box\text{fix}$ ). As  $x \notin \text{FV}_{\geq 1}(\text{fix } z \text{ in box } M)$ , we have that  $x$  does not occur in  $M$ , and thus  $(\text{fix } z \text{ in box } M)[P/x] \equiv \text{fix } z \text{ in box } M$ , and also

$$(\text{box } M[\text{fix } z \text{ in box } M/z])[Q/x] \equiv \text{box } M[\text{fix } z \text{ in box } M/z]$$

Thus, a single use of ( $\Box\text{fix}$ ) suffices.

□

We now pull the following definition out of the hat:

**Definition 2** (Complete development). The *complete development*  $M^*$  of a term  $M$  is defined by the following clauses:

$$\begin{aligned} x^* &\stackrel{\text{def}}{=} x \\ c^* &\stackrel{\text{def}}{=} c \quad (c \in \{\tilde{f}, \hat{n}, \text{zero?}, \dots\}) \\ (\lambda x:A. M)^* &\stackrel{\text{def}}{=} \lambda x:A. M^* \\ (\tilde{f}(\text{box } M))^* &\stackrel{\text{def}}{=} \text{box } f(M) \quad \text{if } M \text{ is closed} \\ ((\lambda x:A. M)N)^* &\stackrel{\text{def}}{=} M^*[N^*/x] \\ (\supset_G \text{ true } M N)^* &\stackrel{\text{def}}{=} M^* \\ (\supset_G \text{ false } M N)^* &\stackrel{\text{def}}{=} N^* \\ (MN)^* &\stackrel{\text{def}}{=} M^*N^* \\ (\text{box } M)^* &\stackrel{\text{def}}{=} \text{box } M \\ (\text{let box } u \Leftarrow \text{box } M \text{ in } N)^* &\stackrel{\text{def}}{=} N^*[M/u] \\ (\text{let box } u \Leftarrow M \text{ in } N)^* &\stackrel{\text{def}}{=} \text{let box } u \Leftarrow M^* \text{ in } N^* \\ (\text{fix } z \text{ in box } M)^* &\stackrel{\text{def}}{=} \text{box } M[\text{fix } z \text{ in box } M/z] \end{aligned}$$



We need the following two technical results as well.

**Lemma 4.**  $M \Longrightarrow M^*$

*Proof.* By induction on the term  $M$ . Most cases follow immediately by (refl), or by the IH and an application of the relevant rule. The case for  $\text{box } M$  follows by (refl), the case for  $\text{fix } z \text{ in box } M$  follows by  $(\Box\text{fix})$ , and the case for  $\tilde{f}(\text{box } M)$  by  $(\Box\text{int})$ .  $\square$

**Lemma 5** (BFV antimonotonicity). *If  $M \Longrightarrow N$  then  $\text{FV}_{\geq 1}(N) \subseteq \text{FV}_{\geq 1}(M)$ .*

*Proof.* By induction on  $M \Longrightarrow N$ .  $\square$

And here is the main result:

**Theorem 7.** *If  $M \Longrightarrow P$ , then  $P \Longrightarrow M^*$ .*

*Proof.* By induction on the generation of  $M \Longrightarrow P$ . The case of (refl) follows by lemma variable rule is trivial, and the cases of congruence rules follow from the IH. We show the rest.

CASE( $\rightarrow \beta$ ). Then we have  $(\lambda x:A. M)N \Longrightarrow M'[N'/x]$ , with  $M \Longrightarrow M'$  and  $N \Longrightarrow N'$ . By the IH,  $M' \Longrightarrow M^*$  and  $N' \Longrightarrow N^*$ . We have that  $x \notin \text{FV}_{\geq 1}(M)$ , so by Lemma 5 we get that  $x \notin \text{FV}_{\geq 1}(M')$ . Hence, by Lemma 3 we get  $M'[N'/x] \Longrightarrow M^*[N^*/x] \equiv ((\lambda x:A. M)N)^*$ .

CASE( $\Box\beta$ ). Then we have

$$\text{let box } u \Leftarrow \text{box } M \text{ in } N \Longrightarrow N'[M/u]$$

where  $N \Longrightarrow N'$ . By the IH,  $N' \Longrightarrow N^*$ , so it follows that

$$N'[M/u] \Longrightarrow N^*[M/u] \equiv (\text{let box } u \Leftarrow \text{box } M \text{ in } N)^*$$

by Lemma 1.

CASE( $\Box\text{fix}$ ). Then we have

$$\text{fix } z \text{ in box } M \Longrightarrow \text{box } M[\text{fix } z \text{ in box } M/z]$$

But  $(\text{fix } z \text{ in box } M)^* \equiv \text{box } M[\text{fix } z \text{ in box } M/z]$ , so a single use of (refl) suffices.

CASE( $\Box\text{int}$ ). Similar.  $\square$